White Rose Research Online

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk
Universities of Leeds, Sheffield and York

Deposited via The University of York.

White Rose Research Online URL for this paper:
https://eprints.whiterose.ac.uk/id/eprint/133819/

Version: Accepted Version

Article:
Ogunyomi, Babajide, Rose, Louis M. and Kolovos, Dimitrios S. (2018) Incremental execution of model-to-text transformations using property access traces. International Journal on Software & Systems Modelling. pp. 1-17. ISSN: 1619-1366

https://doi.org/10.1007/s10270-018-0666-5

Reuse
Items deposited in White Rose Research Online are protected by copyright, with all rights reserved unless indicated otherwise. They may be downloaded and/or printed for private study, or other acts as permitted by national copyright laws. The publisher or other rights holders may allow further reproduction and re-use of the full text version. This is indicated by the licence information on the White Rose Research Online record for the item.

Takedown
If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

UNIVERSITY OF LEEDS
University of Sheffield
UNIVERSITY of York

Incremental Execution of Model-to-Text Transformations using Property Access Traces

Babajide Ogunyomi, Louis M. Rose, and Dimitrios S. Kolovos

Department of Computer Science, University of York
Deramore Lane, Heslington, York, YO10 5GH, UK.
[bjo500, louis.rose, dimitris.kolovos]@york.ac.uk

Abstract. Automatic generation of textual artefacts (including code, documentation, configuration files, build scripts, etc.) from models in a software development process through the application of model-to-text (M2T) transformation is a common MDE activity. Despite the importance of M2T transformation, contemporary M2T languages lack support for developing transformations that scale with the size of the input model. As MDE is applied to systems of increasing size and complexity, a lack of scalability in M2T transformation languages hinders industrial adoption. In this paper, we propose a form of runtime analysis that can be used to identify the impact of source model changes on generated textual artefacts. The structures produced by this runtime analysis, property access traces, can be used to perform efficient source-incremental transformation: our experiments show an average reduction of 60% in transformation execution time compared to non-incremental (batch) transformation.

1 Introduction

Although MDE can reduce systems complexity and increase developer productivity [1], achieving scalability of MDE processes, practices and technologies remains an open research challenge and is important for widespread industrial adoption [2]. The scalability challenges in MDE are numerous, and include: performant persistence of very large models, modularity and reusability in the definition of very large modelling languages, and efficient propagation of changes between artefacts (including models). This paper focuses on the latter challenge, in the context of propagating changes from models to textual artefacts (such as source code, documentation, or build scripts).

While a substantial amount of work has been carried out in the context of incremental model-to-model transformation over the last decade and several prototypes are available, incremental model-to-text transformation has received little attention. Beyond Xpand[1], none of the publicly available model-to-text transformation engines available today support efficient propagation of changes in models to generated textual artefacts.

[1] The incremental Xpand engine is discussed in detail in Section 5.

In the absence of automated incremental model-to-text transformation capabilities, model-driven engineering processes involving large and frequently-changing models grind to a halt unless workarounds are employed. For example, full code generation from large UML models in one of our industrial partners takes hours to complete. To avoid long and wasteful generation cycles, engineers need to steer the generator manually to re-generate only parts of the code that they anticipate to be affected by the changes they have made to the model since the last generation cycle. Performing such impact analysis manually requires intimate knowledge of the code generator and is error prone.

In this paper we demonstrate an approach for achieving fully automated source-incremental model-to-text transformation through the use of *property access traces*. The proposed approach uses runtime analysis to capture information about the way in which a transformation accesses its source models. When the source models change, a property access trace provides an efficient means for determining which subset of the transformation must be re-executed to propagate changes to the textual artefacts. Crucially, a property access trace allows the transformation engine to reduce (and ideally eliminate) execution of the parts of the transformation that are not affected by the changes to the source models, and the M2T transformation scales better as a whole. The proposed approach is agnostic both of the source (modelling) and the target textual language. In Section 4 we discuss the results of experiments we have conducted which show up to a 60% reduction in the average re-execution time of a M2T transformation using the source-incremental technique proposed in this paper.

This paper is an extension to our previous publication [3] in which we proposed a design and prototype implementation for computing and querying property access traces in order to perform efficient propagation of changes from models to textual artefacts[2]. In [3], we demonstrated how an M2T transformation language can be extended with property access traces to incrementally propagate changes in an offline mode. The offline transformation mode assumes that re-synchronisation of generated artefacts with the input model is required only when a new version of the input model is available. In this extension, we expand our investigation to include online transformation. Online transformation enables the synchronisation of generated artefacts with input models on-the-fly, while the input models are being modified (Section 3.5). This paper makes the following additional contributions:

- A new "online" mode for incremental execution of model-to-text transformations that further improves efficiency by leveraging the capability of modern modelling frameworks and tools to provide live model-element-level change notifications (which need to be reconstructed after the fact in the "offline" mode at a significant computational cost)
- An extended empirical evaluation and discussion of the benefits of property access traces for two existing M2T transformations, and comparisons between the online and offline transformation modes (Section 4).

[2] The prototype is available in `https://github.com/epsilonlabs/incremental-egl`.

2 Background

This section briefly summarises contemporary approaches to M2T transformations and the different types of incrementality that are needed for effective and efficient M2T transformation.

The majority of contemporary M2T transformation languages enable transformations to be specified in the form of templates (Listing 1.1), whose structure closely resembles the generated text [4, 5]. Any parts of the generated text that vary over model elements are replaced with *dynamic (executable) sections*, which are evaluated against one or more source models. Any parts of generated text that are not sensitive to the source models are termed *static sections*. A M2T transformation normally comprises several templates, and co-ordination logic that invokes each template on the relevant part of the source models.

```
1  Hello, [%= person.name %]!
```

Listing 1.1. A template-based M2T transformation, in EGL syntax, which contains a static section ("Hello, "), a dynamic section (that outputs the value of the name attribute of a *person* model element) and another static section ("!").

Incrementality in model transformation – and in general – seeks to react to changes in an artefact (such as a model) in a manner that minimises the need for redundant computations. For M2T transformation, three types of incrementality have been identified: user edit-preserving incrementality, target incrementality[3], and source incrementality [5]. User-edit preserving incrementality and target incrementality are now widely supported, but source incrementality is not [6]. In this paper, we focus on source incrementality and argue that it is an essential feature for providing scalable M2T transformation capabilities.

Source incrementality is the capability of an M2T transformation engine to respond to changes in the source models of an M2T transformation in a way that minimises (and ideally eliminates) the need for re-computations that will not eventually have an impact on the latter's output. Achieving a high degree of source incrementality can significantly improve the efficiency of complex transformations, especially when they operate on large or complex source models (e.g., with many cross-references between model elements and/or inter-dependencies between source models).

3 Property Access Traces

In this section, we demonstrate how *property access traces* recorded during the execution of an M2T transformation can be used to detect which templates need to be re-executed in response to a set of changes in the input model(s), thus facilitating source incrementality for contemporary template-based M2T

[3] Target incrementality is achieved when target files are not unnecessarily modified on disk (thus changing their last-modified timestamp) if their content has not changed.

transformation engines. In contrast to existing approaches to source incremental model-to-text and model-to-model transformation, property access traces do not rely on model differencing, which can be computationally expensive and imprecise as discussed in Section 5 where we highlight it as a major drawback of Xpand's[4] incremental transformation approach.

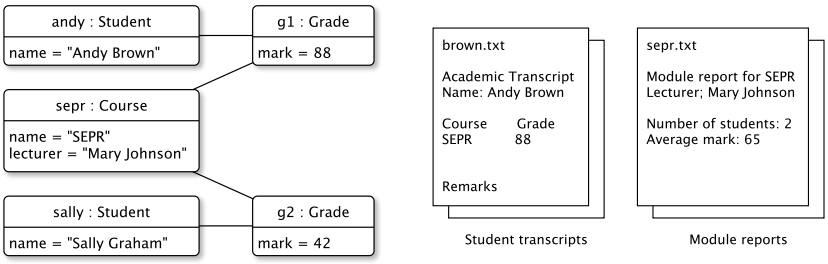
This section provides an overview of using property access traces for source incremental transformation, discusses the way in which existing template-based M2T languages can be extended with support for property access traces, and briefly describes a prototypical implementation of property access traces for the EGL [7] M2T language.

3.1 Overview

To provide support for source incrementality, a transformation engine must be capable of identifying the subset of the transformation that is sensitive to changes in its input models (impact analysis), and re-executing the subset of the transformation to update the target (change propagation). Performing accurate impact analysis presents arguably the greatest challenge: in a template-based M2T transformation, a template might be sensitive to some types of change to a model element, but not to others. In the example presented in Figure 1, student reports are generated by a template that is sensitive to changes to the name of a course (e.g., "SEPR" changes to "Software Project"), but not to the name of the lecturer (e.g., "Mary Johnson" changes to "Mary Johnson-Smith") or to the names of the students that take the module.

Property access traces, as discussed below, provide an effective mechanism for recording an M2T transformation's execution footprint which can be then used to detect relevant changes in the source model, and to determine which parts of the transformation need to be re-executed against which model elements. When a transformation is first executed, property access traces are captured and persisted in non-volatile storage. A *property access trace* records which parts of the transformation access which parts of the source model. In subsequent executions of the transformation, the property access trace is used to detect whether the source model has changed, and to re-execute only those parts of the transformation that are likely to be affected by the source model changes. Determining which parts of the transformation to re-execute is possible because we require that transformation templates have two characteristics: they must be stateless and deterministic. A stateless template takes its data only from the input model, which means that the generated text is dependent only on data that we can observe. A deterministic template is one which when executed twice on the same input performs the same actions and produces the same output, which means that we can always predict which parts of the input model the template will access. Under these conditions, property accesses alone can be used to determine whether or not a re-invocation of a template is likely to produce a

[4] `http://eclipse.org/modeling/m2t/??project=xpand`



(a) Model. (b) Example output.

Fig. 1. Artefacts for an M2T transformation that generates reports.

different output after the input model has been modified. A similar correctness argument is made for the incremental model consistency checking approach in [8].

Property access traces can be applied to a transformation language to provide incrementality in two modes: offline and online. In the offline mode, the transformation engine terminates after an execution, and re-executes a transformation only when a new version of the input model is available. However, it is also possible to apply property access traces to enable immediate change propagation (online transformation). In the online transformation mode, a transformation engine can re-execute a transformation as changes occur in input model(s).

3.2 Design

In order to demonstrate the feasibility of *property access traces*, we extended EGL (the Epsilon Generation Language) [7], a contemporary template-based M2T language. EGX is an orchestration sub-language of EGL which provides mechanisms for co-ordinating template execution. At this point we should stress that although our reference implementation builds on top of EGL, the proposed approach is orthogonal to the model-to-text transformation language and can be in principle implemented on top of any interpreted (e.g. Acceleo[5]) or compiled (e.g. Xtend [9]) language, which can be augmented/instrumented to record property access events. Typically, M2T transformations implemented in compiled languages such as Xtend are likely to be more performant compared to implementations in interpreted languages. However, we have selected EGL as the basis for our prototype largely due to prior familiarity with the internals of the execution engine of the language.

Before discussing the details of implementing *property access* traces for EGL, we first describe the way in which transformations are defined and executed in the language (Figure 2). An M2T transformation in EGL is specified in the form of an EGX program, which comprises a number of rules and EGL templates. In its simplest form, an EGX rule specifies a metamodel type on which it is applicable[6],

[5] `https://www.eclipse.org/acceleo/`

[6] EGX rules also support *guards* which can further limit their applicability

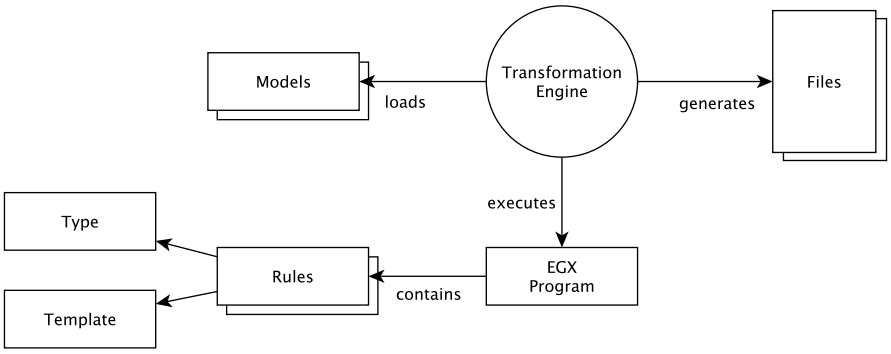


Fig. 2. Overview of transformation execution using EGX.

and expressions that are expected to evaluate to the path of a *template* and the path of a *target* file. The transformation engine starts by loading the input model(s), before executing the EGX program. Then each rule is executed against its applicable model elements; each model element is passed as a parameter to the *template*, and the text produced by the template is stored in *target* file.

Consider, for example, the M2T transformation in Listing 1.2, which produces student transcripts and course reports of the forms shown on the right-hand side of Figure 1. This EGX program comprises two transformation rules: *StudentToTranscript* (lines 1 -5), *CourseToReport* (lines 7 -11). EGX passes each object of type *Student* to the *studentToTranscript.egl* template (Listing 1.3) and each object of type *Course* to the *courseToReport.egl* template (Listing 1.4). Additionally, in each transformation rule, a target (filename) is defined, whose value is determined at the transformation execution time.

In a typical M2T (batch) transformation engine, execution involves evaluating all templates against all instances of that context type every time a transformation is executed. In a source incremental M2T transformation engine, transformation execution involves identifying only the rule-element pairs that need to be re-evaluated to propagate changes from the source model to the generated text. In other words, a source incremental M2T transformation engine identifies but, crucially, does not re-evaluate templates for which the generated text is known from a previous invocation of the transformation.

```
1  rule StudentToTranscript
2     transform student : Student {
3        template : "studentToTranscript.egl"
4        target : student.name + ".txt"
5  }
6
7  rule CourseToReport
8     transform course : Course {
9        template : "courseToReport.egl"
10       target : course.name + ".txt"
11 }
```

Listing 1.2. Example of an EGX M2T program applied to input model in Figure 1(a).

```
1  Student name : [%= student.name %]
2  Course  Grade
3  [% for(grade in student.grades) { %]
4  [%= grade.course.name + " " + grade.mark %]
5  [% } %]
```

Listing 1.3. M2T template for generating student transcripts specified in EGL syntax.

3.3 Extending M2T transformation languages with Property Access Traces.

The implementation of property access traces involves extending the execution engine of an M2T language with four new concepts. During the execution of the transformation, a *PropertyAccessRecorder* captures the properties of the model elements accessed. The recorded *PropertyAccess(es)*, which make up a *PropertyAccessTrace*, are then persisted in non-volatile storage, a *PropertyAccessStore*. Figure 3 illustrates the conceptual organisation of the information contained in a *PropertyAccessTrace*.

- A ***PropertyAccess*** is a triple <e, p, v>, where e is the unique identifier (ID) of the model element, p is the name of the property, and v is the current value of the property. The way in which model element identifiers are computed varies, depending on the underlying modelling technology (e.g., XMI IDs or relative paths for EMF XMI models). Our current implementation does not support monitoring multiple input models, hence, the IDs of the model elements which are globally unique, combined with the name any of a model element's properties are adequate to distinguish property accesses. A *PropertyAccess* is however, trivially extensible to <m, e, p, v> where m is the model name. This would provide a namespacing access and identification mechanism for multiple models and prevents access ambiguity. There are two types of property accesses – *NamedPropertyAccesses* and *FeatureCallAccesses*. *NamedPropertyAccesses* are derived from direct operations on model elements, and they are classified by the type of model element feature (i.e., *AttributeAccesses* and *ReferenceAccesses*) that is accessed and

```
1  Course Report for [%= course.name %]
2  Lecturer: [%= course.lecturer %]
3
4  Number of students:[%= course.grades.size() %]
5  Average mark:[%=course.grades.collect(mark).sum()
6                 /course.grades.size() %]
```

Listing 1.4. M2T template for generating course reports specified in EGL syntax

the type of value that they store. *AttributeAccesses* store a string value and are used when the accessed property type is a primitive. An *AttributeAccess* is derived when a model element's feature is accessed in a template (e.g., the execution of this statement: *person.name*). *ReferenceAccesses* store the unique identifiers of the referenced model elements and are obtained when the feature of a model element that is accessed is an association between two classes or between two instances of a class (e.g., person.followers). Lastly, a *FeatureCallAccess* is derived from expressions that access instances of a metamodel type, rather than properties of a model element (e.g., Person.all, Person.allInstances). *FeatureCallAccesses* return a collection that contains the unique identifiers of the objects.

- A ***PropertyAccessTrace*** (Figure 3) captures which transformation rules are invoked on which source model elements and, moreover, which *PropertyAccesses* resulted from each invocation of a transformation rule (a *RuleInvocation* in Figure 3).
- A ***PropertyAccessRecorder*** is responsible for recording *PropertyAccesses* during the execution of a template, and updating the *PropertyAccesses* when a change in the value of a *PropertyAccess* is detected. It is important to note that since property access traces contains data about input model elements only, any other type of change to the transformation specification is not considered (See section 4.3 for a discussion on known limitations of this approach).
- A ***PropertyAccessStore*** is responsible for storing the *PropertyAccesses* passed on to it by the *PropertyAccessRecorder*. The *PropertyAccessStore* is also responsible for making *PropertyAccesses* (that were stored during a previous transformation execution) available to the transformation engine. In our prototype, we use an embedded RDBMS to store *property accesses*, but other options (e.g., graph databases, XML documents, etc.) are also possible. The main non-functional requirement for a *PropertyAccessStore* is performance: any gains achieved with a source incremental engine might be negated if the *PropertyAccessStore* cannot efficiently read and write property access traces.

We now briefly describe the way in which these concepts are used to achieve source incremental transformation, before providing an example. During the initial execution of a transformation, the *PropertyAccessRecorder* creates *PropertyAccesses* from the properties of model elements that are accessed during the

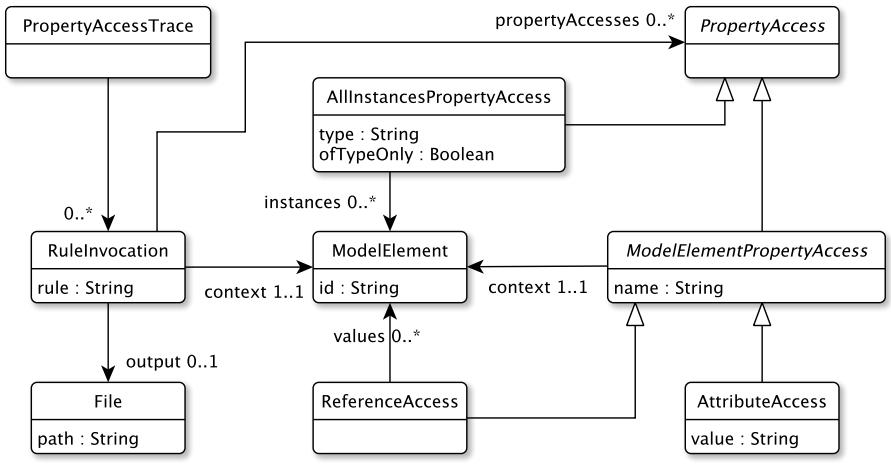


Fig. 3. Overview of Property Access Trace.

execution of each rule. The collected *PropertyAccesses* are organised by *RuleInvocation* by the transformation engine to form a *PropertyAccessTrace*, that is eventually stored by the *PropertyAccessStore*. In a subsequent execution of the M2T transformation, the transformation engine retrieves the previous *PropertyAccessTrace* from the *PropertyAccessStore*. Whenever the transformation engine would ordinarily invoke a transformation rule, it instead retrieves each relevant *PropertyAccess* from the *PropertyAccessTrace* and queries the model to determine if the value of any of the *PropertyAccess*es has changed. Only when a value has changed is the transformation rule invoked. The *PropertyAccessTrace* is updated and stored if any values have changed.

3.4 Offline Transformation in EGL

In the offline transformation mode, the transformation is only re-executed on demand when a new version of an input model is available. During the initial execution of a transformation, the transformation engine records and persists property accesses. During subsequent executions of the transformation, the transformation engine retrieves the persisted property access trace, and before re-executing a template on a model element, it examines the values of associated property accesses that were retrieved from the property access store by querying the input model to determine whether there has been any modification of any of the model element features that were accessed during the previous execution of the template. If the current value of at least one template invocation's property access differs from the retrieved value of such property access, only then will the transformation engine re-execute the template. It is important to note that templates often contain multiple accesses to the same model element feature.

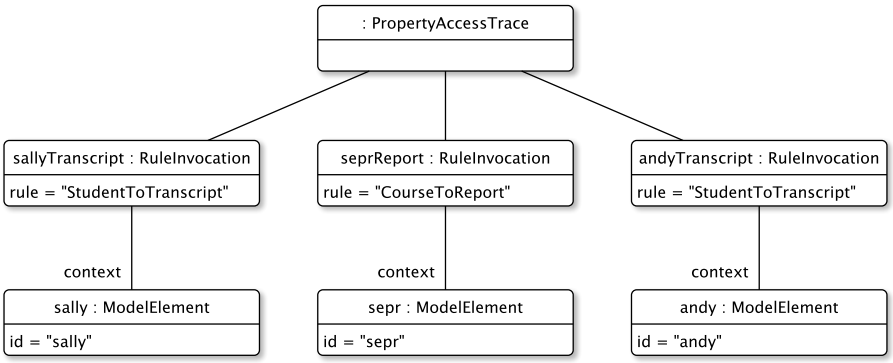


Fig. 4. A partial property access trace for executing *studentToTranscript.egl* on *andy* and *sally*, and *courseToReport.egl* on *sepr*.

Therefore, in order to minimize the space requirements for persisting a property access trace, only distinct property accesses are recorded.

3.4.1 Offline Transformation Example To further demonstrate the way in which property access traces achieve source-incremental M2T transformation, we now consider an example. Our example uses the transformation in Listings 1.3 and 1.4, which generate student transcripts and course reports from a university model. Executing the transformation on the minimal university model in Figure 1(a) causes the transcript-generating rule to be invoked once on each student (*andy* and *sally*), and the course report-generating rule once on course (*sepr*). As such, the resulting property access trace comprises three rule invocation objects (Figure 4). Each rule invocation object comprises several property accesses, which are recorded during the execution of the templates in Listing 1.3 and 1.4.

Let us consider the properties accessed during the invocation of the template on *sally*. The *sallyTranscript* rule invocation (Figure 5) comprises several attribute and reference access objects and is constructed as follows. Firstly, the template accesses `sally.name` (line 1 of Listing 1.3) and creates the *aa1* attribute access (Figure 5). The template then accesses `sally.grades` (line 3) and this creates the *ra1* reference access. The *grade.course.name* traversal expression in the template (line 4) creates two property accesses: the *ra2* reference access for *grade.course* and the *aa2* attribute access for *course.name*. Finally, the *grade.mark* expression (line 5) creates the *aa3* attribute access. The boxes with a dashed border in Figure 5 reinforce the relationship between property access objects in the trace and the expressions in the template (Listing 1.3). Note that each property access stores a reference to the model element from which its value was obtained.

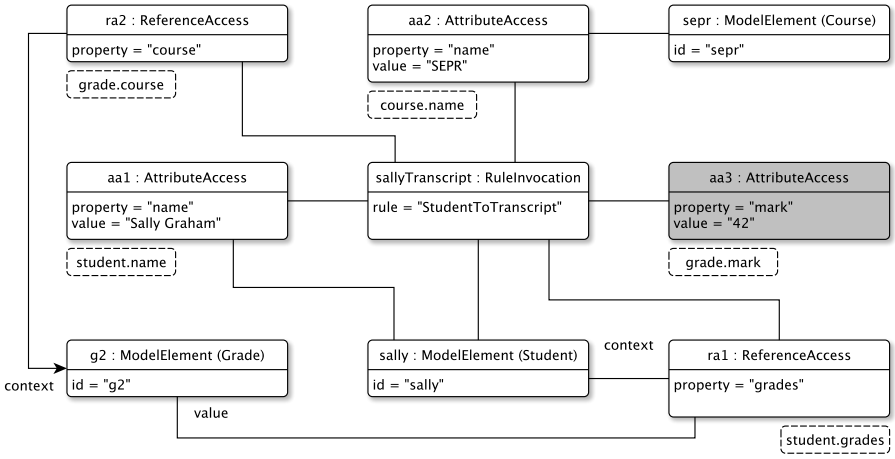


Fig. 5. Expansion of the property access trace for the *sallyTranscript* rule invocation.

When the M2T transformation is executed again, the transformation engine retrieves the property access trace (including Figures 4 and 5) and queries the parts of the model that were previously accessed by the transformation, such as the name of each student. Only when the value of any property differs from the value stored in a property access is the containing rule invocation re-executed.

For example, the *sallyTranscript* rule invocation (Figure 5) indicates that if all of the following constraints hold, then the rule invocation need not be re-executed:

1. `sally.name == "Sally Graham"` – due to *aa1*
2. `sally.grades == {g2}` – due to *ra1*
3. `g2.course == sepr` – due to *ra2*
4. `sepr.name == "SEPR"` – due to *aa2*
5. `g2.mark == "42"` – due to *aa3*

Suppose that Sally's grade for the SEPR course is changed: the mark attribute of *g2* is changed from 42 to 54. Note that the *aa3* attribute access (highlighted in Figure 5) stores the old value for the mark, 42. When the transformation is re-executed, condition #5 above will no longer hold: *g2.mark* will now evaluate to 54. Consequently, the transformation engine will re-execute the *sallyTranscript* rule invocation.

We have not shown the complete property access trace for the *andyTranscript* rule invocation (due to space constraints), but it is very similar in structure

to the *sallyTranscript* rule invocation in Figure 5. The property accesses for *andyTranscript* result in the following constraints:

1. `andy.name == "Andy Brown"`
2. `andy.grades == {g1}`
3. `g1.course == sepr`
4. `sepr.name == "SEPR"`
5. `g1.mark == "88"`

From these constraints, it is clear that the change to *g2.mark* does not require a re-execution of the *andyTranscript* rule invocation as none of the constraints above depend on *g2.mark*. If, on the other hand, our change had been to *sepr.name* rather than to *g2.mark*, then both of the sets of constraints shown above would be unsatisfied and both the *sallyTranscript* and the *andyTranscript* rule invocations would be re-executed. In addition to the simple types of modifications outlined above, deleting obsolete files (generated files that exist in the transformation output directory despite having deleted the model elements from which they were generated) from transformation output directories is straightforward. The transformation engine evaluates whether previous rule invocations are still valid by checking if the referenced model element of its associated template invocations is still accessible in the source model. If a model element has been deleted, then any associated rule invocation will no longer be executable, and hence the file generated from the previous rule invocation is deleted.

In general, determining whether or not a rule invocation needs to be re-executed requires the evaluation of *O(n)* constraints where n is the number of distinct property accesses for that rule invocation.

3.5 Online Transformation in EGL

In this section, we discuss how property access traces can be extended to support on-the-fly (or online) incremental M2T transformation. Thus far, the narrative has assumed that M2T transformations are only re-executed when a new version of an input model is available. The example in Section 3.4.1 describes a typical offline incremental M2T transformation scenario – initially, the transformation is executed and some textual artefacts are generated; the input model evolves and results in a new version of the model; the transformation is re-executed, and generated artefacts are re-synchronised with the input model. In the offline M2T transformation we assume that the model editing process happens in a black box, and we can only have access to the latest version of the transformation's input model. However, some modelling frameworks and tools (e.g. EMF, PTC Integrity Modeler) offer model-element-level change notification facilities which can be leveraged to eliminate the need for post-fact change detection and facilitate online incremental M2T transformation. Online transformation can be particularly important for modelling tools which require instant propagation of model changes to generated artefacts to immediately evaluate the effects of the changes and ensure consistency.

Commonly, modelling tools that are used to construct graphical editors for modelling languages (e.g., Eugenia [10]) perform M2T transformations which generate source code (e.g., implementation classes that adapt the model classes for editing) for the editor. For example, each time an input model is modified, Eugenia updates an intermediate model (*genmodel*) through a M2M transformation before re-executing the M2T transformation that generates the editor from the intermediate model. As the construction of a model editor involves several iterations which contain minor tweaks, immediate re-synchronisation of the editor code with the *genmodel* is desirable, and can potentially reduce development time, since the developer can instantly assess the effects of the changes on the model editor. Another important advantage of online change propagation it is arguably more time-efficient than the offline transformation mode because the transformation engine does not need to fully traverse the input model to detect changes; fine-grained change notifications are provided for free by the model editor.

3.5.1 Design Although the online and offline incremental transformation modes are founded on the same principle of recording property accesses on model elements during template invocations and using the recorded data to limit the re-execution of a transformation to relevant template invocations, noteworthy differences also exist between the two modes.

1. In the offline mode change detection is performed by the transformation engine in a batch mode, whereas in the online mode, the transformation engine is notified of model-element-level changes;
2. Impact analysis in the offline mode requires full model and property access trace traversals, while in the online mode, only a subset of the property access trace is examined. As such, the offline mode requires the evaluation of *O(n)* constraints where n is the number of property access traces. On the other hand, during impact analysis in the online mode, the transformation engine only analyzes the stored property access traces, which – using a Java HashMap – is an *O(1)* operation in the average case.

In the offline mode, change detection entails querying the input model to determine whether the values of relevant property accesses have changed. In the online mode, change notifications are provided by the underlying model editor. It follows that in the online mode, a property access does not need to include the value of the property since the transformation does not require this information to determine whether the value of a model element's feature has been modified or not. Therefore, a property access becomes a pair of the form $< e, p >$, where e is the model element id, p is the model element's property name. Although change detection in the online mode is relatively straightforward compared to the offline mode, the online mode introduces another layer of complexity – establishing transaction boundaries. For instance, the transformation engine would have to determine when to re-execute a transformation depending on whether an edit session of the input model is still in progress or has completed. Also, careful

considerations have to be made of how the transformation engine responds to change notifications it receives while a previous invocation of a transformation is still in progress.

Furthermore, online transformation introduces the notion that model element changes are external and an indirect concern of the transformation engine since the transformation engine does not have to compute them. Therefore, model element change is conceptualized as an external entity that is consumed by the transformation engine much like a transformation parameter. Hence, in addition to *PropertyAccessTrace*, *PropertyAccessRecorder*, *PropertyAccess*, and *PropertyAccessStore*, a further concept is introduced:

- A ***Change*** comprises a model element's id and the name of the property of the modified model element (i.e., a pair of the form $< e,\ p >$) and it is structurally equivalent to a property access. It is used to determine which rule invocations require re-execution.

An overview of online transformation using property access traces is presented in Figure 6. In the online propagation mode, the execution of a transformation proceeds as follows:

1. The transformation engine begins to observe the input model specified in the transformation configuration.
2. The model editor triggers change notifications as the user edits the model.
3. The transformation engine receives change notifications for an input model as they occur.
4. The transformation engine analyses the change notifications to find relevant rule invocations.
5. The transformation engine re-executes related rule invocations, executes new rule invocations (for new elements), and deletes previously generated obsolete files (for deleted elements).
6. Based on step 5, any changes to previously recorded property accesses are updated and persisted in the property access store.

Since the transformation engine needs to be aware of changes as they are made to the input model, in step 1, before the initial execution of a transformation, the transformation engine initiates monitoring of the model elements through a change listener (provided by the underlying modelling framework) to the input model. In step 2, modifications to observed model elements will trigger change notifications by the underlying modelling framework which are forwarded to the transformation engine (in step 3). In step 4, the transformation queries the property access trace for each change notification to determine which rule invocations are affected by the change. In step 5 new rule invocations are created (for new elements), affected rule invocations are re-executed, and obsolete files are deleted. Finally, in step 6, the property access trace is updated and persisted in the property access store.

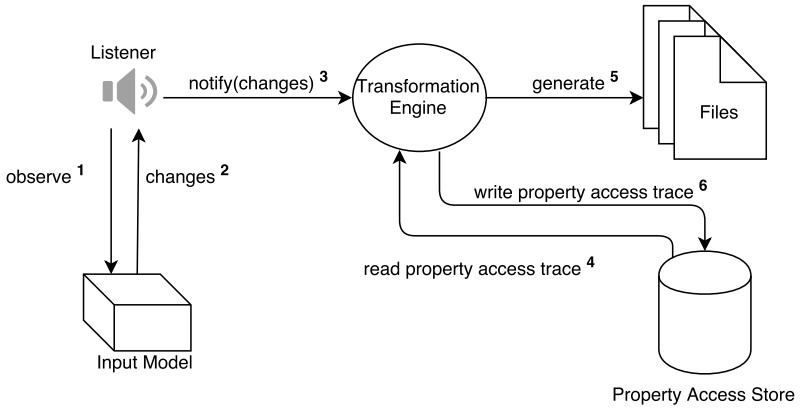


Fig. 6. Overview of Online transformation using Property Access Trace.

3.5.2 Change Detection and Batching in Online Transformation As explained above, to achieve online incremental transformation using property access traces, it is essential that the tool used to edit the source model is capable of producing live model-element level change notifications.

With regard to the frequency of incremental re-execution in response to such notifications, the transformation engine can either react to each change as it occurs, and re-execute the appropriate rule-invocations immediately, or it can collect and batch changes based on an appropriate policy. Potential policies can include processing changes every time the user saves a model file, at fixed intervals, after change events that leave the model in a state where it satisfies its constraints, or using a combination of the above. A discussion on the appropriateness of different policies for different use-cases is out of the scope of the paper. As such, our only assumption is that changes will be processed in batches (in the case of immediate re-execution, each batch contains exactly one change). To minimise rule invocations the following types of change notifications can be safely filtered out from a batch before passing it on to the impact analysis algorithm:

- **Duplicate property change events:** The impact analysis algorithm only needs to know which (distinct) properties of model elements have been modified during the last execution of the transformation, but is insensitive to the *number* of times the same property value may have been changed. As such, we only keep one change notification for each distinct model element-property pair.
- **Property change events on elements deleted later on in the editing session:** Re-generating the content of target files only to delete them altogether later on when the deletion event is encountered is wasteful and is preempted by ignoring property change events from elements that are deleted subsequently in the editing session.

– **Property change events on elements created in the editing session:** Properties of new model elements cannot have been possibly accessed in previous execution of the transformation, and as such changes in their properties are inconsequential for the purposes of impact analysis.
– **Deletion events for elements that were created in the editing session:** The deletion of model elements created in the same editing session is also irrelevant to impact analysis, in line with the argument above.

As our prototype implementation is based on EMF, we also filter out EMF-specific change notifications that are fundamentally irrelevant to impact analysis (e.g. EMF produces a notification every time a listener is added or removed to a model element). The filtered changes are then used to conduct impact analysis and selectively re-execute relevant rules identically to the offline mode.

4 Evaluation and Experience Report

In this section we report on the results of the empirical evaluation of the proposed approach, in which we compare the transformation execution times in incremental (offline and online modes) and non-incremental execution modes for two existing transformations. The results of our experiments show that source incremental transformations can be more efficient than non-incremental transformations, particularly for frequent or relatively small changes to models.

4.1 Empirical Evaluation

To assess the performance of *property access traces*, we used two existing EGL transformations: Pongo and INESS. We investigated whether property access traces are effective when used for repeated invocations of M2T over the lifetime of an MDE project (Pongo). In order to demonstrate that property access traces scale well with complex M2T transformations, we performed an experiment with the INESS transformation. We also investigated the memory and disk usage of *property access traces* (Pongo and INESS) to ensure that resource usage is reasonable. The first set of Pongo experiments (Section 4.1.1) was executed in the offline mode, and the second set (Section 4.1.2) was executed in the online mode. Each experiment was run a total of ten times, after which we computed the average execution time for each iteration. The experiments that were performed during the evaluation work were executed on a MacBook Pro OS X Yosemite (2.5 GHz Intel Core i5, 8 GB 1600 MHz DDR3) with Java 1.7 SDK.

4.1.1 Pongo Experiment I Pongo[7] generates data mapper layers for MongoDB, a non-relational database. Pongo takes as input an Ecore model that describes the types and properties of the objects to be stored in the database, and generates Java code that can be used to interact with the database via the user-defined types and properties (without needing to use the MongoDB API).

[7] `https://github.com/kolovos/pongo`

We compared the total time taken for incremental and non-incremental code generation over the lifetime of a real MDE project. For this purpose we used Pongo v0.5, and 11 versions of the GmfGraph Ecore model (obtained from the Git repository[8] of the GMF team). To simulate code generation activities in the GMF project, we ran Pongo using non-incremental and incremental EGL in offline mode on each version of the GmfGraph Ecore model.

The results (Table 1) show the difference in number of template invocations and total execution time between non-incremental and incremental execution modes of execution, for each of the 11 versions of the GmfGraph model. Expectedly, during the first invocation of the transformation (version 1.23) in incremental mode, the execution took slightly longer to execute than the non-incremental mode because the former incurs an overhead as the transformation in addition to evaluating templates, must record and process model element properties that are accessed in each template. However, during subsequent executions of the transformation, the incremental mode of execution required between 25% and 48% of the execution time required by the non-incremental mode. In other words, during the execution of the transformation on all versions of the GmfGraph project, we observed upto an 75% reduction in total execution time. Although the overall reduction in execution time (12.51s) is modest, that is partly explained by the relatively small size of the Pongo transformation (6 EGL templates totalling 329 lines of code), and of the GmfGraph model (averaging 65 classes).

		Non-Incremental		**Incremental**	
Version	Elements Changed (#)	Invocations (#)	Time (s)	Invocations (#)	Time (s; %)
1.23	-	72	1.79	72	2.29 (128%)
1.24	1	73	1.72	6	0.49 (28%)
1.25	1	73	2.01	2	0.50 (25%)
1.26	1	74	2.03	6	0.53 (26%)
1.27	10	74	1.97	44	0.95 (48%)
1.28	10	74	1.95	44	0.93 (48%)
1.29	14	74	1.94	14	0.56 (29%)
1.30	24	77	2.02	38	0.94 (47%)
1.31	1	77	1.86	0	0.49 (26%)
1.32	1	77	1.95	0	0.48 (25%)
1.33	3	79	2.00	8	0.57 (29%)
			21.24		8.73 (41%)

Table 1. Results of using non-incremental and offline property access traces for incremental M2T transformation for the Pongo M2T transformation, applied to 11 historical versions of the GmfGraph Ecore model.

4.1.2 Pongo Experiment II

To evaluate property access traces in the online transformation mode using the Pongo M2T, we first had to re-construct the changes that existed in-between the 11 versions of GmfGraph. In the first step,

[8] `https://git.eclipse.org/c/gmf-tooling`

we determined the differences between an input model and its preceding version. This was done using the EMFCompare tool [11] which computes a *difference* model by comparing two models. EMFCompare has a graphical user interface which shows the differences between two models in both textual and tree view formats. In the next step, we manually encoded the changes identified via model differencing as a script that could be used to evolve a model from its current version to the next version.

Considering that our experiment replays the evolution of the input models, it is important to note that during this process, the changes between the versions of the input model are batched before being forwarded to the transformation engine (as depicted in Figure 7). This is based on the assumption that during the evolution of each version of the input model, the changes were applied in a single modification session because this information is not stored in either version of the model. Moreover, it is impossible to determine exactly what changes were done and undone during the real evolution of the input model.

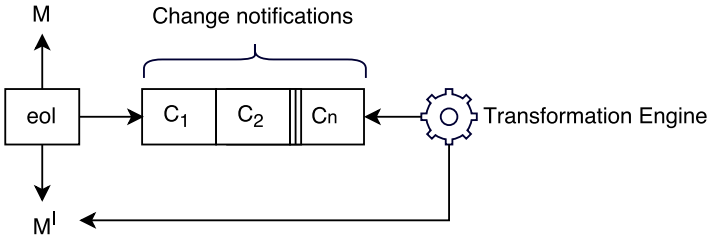


Fig. 7. Overview of online execution of Pongo on GmfGraph.

		Non-Incremental		**Incremental (Offline)**		**online**	
Version	Changes (#)	Inv. (#)	Time (s)	Inv. (#)	Time (s; %)	Inv. (#)	Time (s; %)
1.23	-	72	1.79	72	2.29 (128%)	72	2.42 (135%)
1.24	1	73	1.72	6	0.49 (28%)	6	0.14 (8%)
1.25	1	73	2.01	2	0.50 (25%)	2	0.06 (3%)
1.26	1	74	2.03	6	0.53 (26%)	6	0.13 (6%)
1.27	10	74	1.97	44	0.95 (48%)	44	0.65 (33%)
1.28	10	74	1.95	44	0.93 (48%)	44	0.63 (32%)
1.29	14	74	1.94	14	0.56 (29%)	14	0.20 (10%)
1.30	24	77	2.02	38	0.94 (47%)	38	0.79 (39%)
1.31	1	77	1.86	0	0.49 (26%)	0	0.03 (1%)
1.32	1	77	1.95	0	0.48 (25%)	0	0.02 (1%)
1.33	3	79	2.00	8	0.57 (29%)	8	0.24 (12%)
			21.24		8.73 (41%)		5.31 (25%)

Table 2. Comparison of a non-incremental, incremental transformation using property access traces in offline and online modes for the Pongo M2T transformation, applied to 11 historical versions of the GmfGraph Ecore model. (Inv. refers to invocations)

Table 2 compares the execution times of the Pongo transformation on GmfGraph models in non-incremental and incremental modes (online and offline executions). Overall, as expected, incremental execution required less time compared to the non-incremental execution, and online incremental execution outperformed offline incremental execution. The difference in execution time between the online and offline execution modes is due to the fact that the transformation engine in the online mode does not need to compute model changes by querying the input model, as these are provided for free by the EMF notification facility. By contrast, in the offline mode the transformation engine performs a full traversal of the input model and analyzes an entire stored property access trace in order to determine template invocations that require re-execution. As highlighted in Table 2 during the evolution of the model from version 1.30 to 1.31, and 1.31 to 1.32, which comprised single changes to the input models, the transformation engine in offline mode still took approximately 0.5 seconds despite the fact that the changes were irrelevant to the transformation (i.e., did not result in the re-execution of any template invocation). On the other hand, the online mode took about 0.02 seconds to re-execute (significantly less time compared to 0.48 seconds in offline mode).

4.1.3 INESS Experiment For this experiment we used an existing transformation developed in the context of the Integrated European Signaling System (INESS) EC FP7 project. The INESS M2T transformation generates model checking code (in Promela [12] and mCRL2 [13]) for analysing railway interlocking models (captured in UML) to determine inconsistencies between requirements and system properties defined by railway engineers. For this experiment, we only considered the subset of INESS M2T transformation that generated mCRL2 code because it represents a bottleneck of the transformation. It is also a complex part of the transformation: the input model from which mCRL2 code was generated was about 20 MB and contained 119,621 model elements. Additionally, the transformation comprised 6 templates. This experiment was executed in offline mode and in five iterations as follows:

1. Start with a version M of the UML model
2. Run the transformation on M in incremental mode
3. For each iteration
   (a) Manually modify M and obtain M'
   (b) Run the transformation on M' in non-incremental mode and record its execution time
   (c) Run the transformation on M' in incremental mode and record its execution time

The model (M) we started with in Step 1 was the 20MB UML model discussed above. In Step 2 we executed the transformation in incremental mode to establish the property access traces. Then we conducted five iterations of Step 3. The first step of each iteration (3a) involved introducing small-scale changes to the model (M) in order to obtain a new version of it (M'). The specific size and

nature of these modifications are not important as our sole purpose with these was to trigger controlled re-execution of templates in the incremental mode. Then in Steps 3b and 3c we executed the transformation again on the modified model (M') in incremental and non-incremental mode and recorded the respective execution times, which are displayed in Table 3.

		Non-Incremental		**Incremental**	
Iteration	Elements Changed (#)	Invocations (#)	Time (s)	Invocations (#)	Time (s; %)
1	-	4	25 072	4	50 218 (200%)
2	1	4	25 511	1	5.23 (0.02%)
3	2	4	24 122	2	10.77 (0.04%)
4	3	4	24 901	3	11.10 (0.04%)
5	4	4	25 227	4	51 017 (202%)

Table 3. Results of using property access traces for offline incremental M2T transformation of INESS M2T transformation compared to the non-incremental execution of the transformation.

The obtained results provide further evidence that property access traces can be used to reduce the amount of time required to propagate model changes to generated artefacts. Apart from this, the results also provide interesting insight into the nature of the INESS transformation; the amount of time required to re-execute the transformation in incremental mode during iteration 5 is 200% of the time expended re-executing the transformation during iteration 4. Considering that only one additional file is re-generated during iteration 5 compared to iteration 4, there is a disparity in the re-execution time observed during iterations 4 and 5. This disparity is due to the fact that the INESS transformation includes a long-running, monolithic template (mCRL2_procs_v2.egl) from which a large file was generated. The re-execution of the *mCRL2_procs_v2.egl* template accounted for more than 99% of the total execution time of the transformation. Precisely, it required about 14 hours to execute in incremental mode, and generated a file that is about 25 MB on disk[9]. This observation motivates further work discussed in Section 6.

As the main purpose of the INESS M2T experiment was to demonstrate that property access traces are tractable with respect to space and memory usage for M2T transformations of high complexity (e.g., transformations that consume large input models), we describe our observations during the experiment in terms of memory and disk space utilization. As summarized in Table 4, a total of 88,011 unique property accesses due to access operations made on 32,117 model elements, were recorded during the execution of the transformation. This figure indicates that about 27% of the input model elements are accessed while about 73% (proportion of model element features) of the input model is not accessed

[9] EGL uses efficient internal representation mechanisms (i.e. string buffers) and standard file I/O operations. The bulk of this time is spent on genuine execution of the logic of the template.

by the transformation. Note that this figure remained unchanged throughout the five iterations because the input model modifications did not include deletions of model elements.

Input model			Property Access Trace			
Size (MB)	Elements (#)	Features (#)	Size (MB)	Elements (#)	PA (#)	Memory (MB)
20	119 621	481 147	19.7	32 117 (27%)	88 011 (18%)	4.22

Table 4. Summary of the space requirements for the INESS transformation.

4.1.4 Memory and Disk Utilization To demonstrate that our approach is practical with respect to resource usage, we investigated the memory and disk usage of property access traces during our experiments with Pongo and INESS. In the Pongo M2T experiments on GmfGraph models, the average number of property accesses was 797 which represents about 38 KB in memory cost while disk space requirement for persisting the property access traces was 252 KB in offline mode. In contrast, in online mode, the average disk space consumption of the execution of Pongo M2T on GmfGraph models was 236 KB. On the other hand, the average memory utilization recorded during the execution of INESS M2T was 4.22 MB, and the property access trace was persisted in a database file which occupied about 19.7 MB of disk space.

It is important to note that the memory and disk usage will vary for different transformations, depending on the size of the input model and in particular the amount of *property accesses* made by the transformation.

4.2 Discussion

Our experiments suggested that property access traces improve the scalability of M2T transformations. From our results, the time required to incrementally propagate changes from input models to generated artefacts depended on the impact of the changes made to the input models. Through our experiments on real life M2T transformations, we have demonstrated reduction in execution time for both contrived/controlled (e.g. the changes to the UML model in the INESS case study) and real-world changes to a model (e.g., the changes made to GmfGraph Ecore model which we were able to reproduce using EOL scripts). The results also indicate that source incrementality using our approach is more efficient than non-incremental transformations when frequent, small changes are made to a model throughout the lifetime of a project.

The results of the experiments in our previous work [14], which used *signatures* for source-incremental M2T transformation suggested that source incrementality can be used to realize up to 45% performance gain in transformation execution time. As observed during our experiments, *Property access traces* in the offline mode offer a further 15% reduction in transformation execution time.

Overall, a 60% reduction in transformation execution time was observed using *property access* traces in the offline mode. In the online mode, overall, 75% reduction in transformation execution time was observed.

As demonstrated in Section 4.1.3, the benefits of incremental change propagation largely disappear for transformations that produce large monolithic files. Such transformations can be the result of (1) a poor implementation or (2) missing cross-file referencing (e.g. *import* statements) facilities in the target language (as is the case with Promela). The first case is out of the scope of this work. In our experience, the second case is rather infrequent as the overwhelming majority of programming and markup languages provide support for cross-file references.

Lastly, an incremental M2T transformation is correct if it results in the regeneration of all the required files whose contents were affected by the change(s) to the input model. Ideally, this can proved by formal verification. However, to the best of our knowledge, most contemporary M2T languages (including EGL, Acceleo, Xtend) do not have formally specified semantics, and as such are not amenable to formal proofs. Hence, to build confidence on the correctness of the incremental execution of the two transformations that we used in the evaluation of *property access traces*, we performed a substantial number of tests which compared the output of the transformations in incremental mode with the output of the transformations in non-incremental mode. The outcome of our tests indicate that the contents of the files generated in incremental mode were always the same as the contents of the same files generated in non-incremental mode.

4.3 Limitations of Property Access Traces

Our current implementation of *property access traces* in EGL monitors property accesses only during the execution of templates. However, in the offline transformation execution mode, *property access traces* can become over-sensitive to changes to parameters contained in unordered collections because it cannot distinguish between unordered and ordered collections. Consider a template (e.g., `[%= student.grades.mark %]`) that only prints out the grades of a student, the *PropertyAccessRecorder* records a property access of *grades* on *student*, whose value is a collection of *Grade*s, and also records a property access of *mark* on each *Grade* in the collection *Student.grades*. If in a change event, a *Grade* is removed and re-added to the collection *Student.grades*, these modification operations will result in the same set of *Student.grades*, albeit with a different order, since the re-added *Grade* is inserted at the back of the collection. In the offline mode, this will cause the template to be re-executed unnecessarily. The order of collections are important for accurate comparison of modified structural features of a model element. Our current implementation does a string comparison of the values of *property accesses* recorded from calls that return a collection of structural features, and cannot detect if mere re-ordering of collections is a significant change event.

An inherent limitation of the *property access trace* approach is that the use of non-deterministic programming constructs (e.g., random number generators,

hash-sets, hash-maps) in a template prevents source incrementality (because the template is not guaranteed to produce the same result in two subsequent invocations even if the model has not changed between them), and that property access traces can be pessimistic: it is conceivable that a template might access a property but not use its value in the generated text (e.g., `[% if(grade.mark > 70) { //do-nothing } %]`). In the latter case, a property access trace would result in an unnecessary re-execution of the template.

Finally, an implementation-level limitation of the current prototype is that it does not support multiple input models. However, this limitation is trivial and can be easily addressed by extending the *property access trace* as discussed in Section 3.2. Although the case studies presented in this paper considered only M2T transformations that comprised single input models, the performance of property access traces is not likely to diminish compared to what our current experiments suggest, if multiple input models were otherwise used.

The reason for this is that property access traces scale by the number of element-property pairs accessed during a transformation and require no further information from the models where these elements are contained, other than a unique (in the scope of the transformation) model identifier.

5 Related Work

Available literature indicates that significantly more research work has been done in the context of incremental M2M transformation. Hearnden [15] proposes a live-updating technique which uses a tree to represent the trace of a transformation execution. Each node represents either a source or target element, and the edges are rules. The transformation context exists as a whole and is maintained throughout all transformation executions. Thus, as changes are made to the source model, the changes reflect in the tree and the target is deduced from the tree. For example if an element is added to the source, the transformation searches for a matching rule and model element in the tree. If it finds one, it updates the node, otherwise, it spawns another edge and node in the tree. This way, the transformation eliminates the need of a merge algorithm for merging incremental targets.

Other techniques described by Giese et.al. [16] and Ráth et. al. [17] are graph-based. Giese et.al.'s technique is based on Triple Graph Grammars (TGG) and exploits persisted traceability information to maintain consistency between source and target models; the correspondence model has a correspondence node with a self-association which connects each correspondence node to its predecessors. A rule in TGG specifies a correspondence mapping between the elements of the source and the target models. A graph grammar rule is applied by substituting the left hand side (LHS) with the right hand side (RHS) if the pattern on the LHS can be matched to a pattern in the correspondence model. A directed edge from the correspondence node to the created target element is inserted each time a rule is successfully applied. This reflects the dependencies and execution order of the rules. So, by traversing the directed acyclic graph created by the

correspondence nodes, inconsistencies between the source and target models can be determined, which is done by retrieving the rule which was used to create the correspondence node and checking if it still matches the current situation. For example, if any inconsistency is detected due to a deletion in the source, it deletes the created target element and the correspondence node. This way, the algorithm achieves incrementality by not re-running the transformation against the entire source model but it incurs a cost in all correspondence nodes by comparing it with patterns in the source model.

A similar TGG-based incremental transformation technique is proposed by Giese et. al [18]. This technique requires TGG rules to be deterministic and assumes only source model modification. In contrast to [16], it supports live change propagation. To detect model element modifications, an event listener is attached to each model element in the source. Whenever an element is modified, its associated correspondence node is put in a transformation queue. Thereafter, the queue is processed by executing specific synchronization rules on the elements contained in the queue. Synchronization rules are responsible for maintaining consistency between associated source and target models by first checking the structure of the matching source and target elements before checking attribute equality.

A different incremental graph-based transformation technique called incremental pattern matching is presented by Ráth et. al. [17]. Graph patterns are atomic units of model transformations which define constraints that must be satisfied for model manipulation to take place. In case of incremental pattern matching, graph patterns are defined on model elements, and whenever the model is modified, graph patterns are updated. This approach is based on the RETE algorithm. RETE is an efficient algorithm for comparing large collections of patterns to collections of objects [19]. Transformation information is represented as tuples and nodes. Tuples contain model elements. Nodes refer to patterns and store tuples that conform to a pattern. When changes are applied to a model element, update signals are sent through outgoing edges, then each receiving node updates its stored tuples, and where applicable, more update signals are generated and propagated through the graph.

Bergmann et. al. [20] presents a fundamentally different approach which defines the concept of *change history models*. Change history models are essentially a log of elementary model changes derived from performing simple operations on model elements. Basically, the change history models are are trace models which contain sequences of model manipulation operations. So, whenever a change is applied to the model, the change history model is updated and the associated rule is invoked. Since this technique focuses on bi-directional transformation (i.e., transformation in which target elements can also be translated to source elements), a change history model of the target model is also tracked. Incrementality is achieved by mapping the change history model of the source and target models based on a generic metamodel that captures model manipulation operations, e.g., *createElement, setAttribute*. A distinct feature of this technique is that transformation mapping takes place between model manipulation opera-

tions in the change history model of both source and target models rather than on the source and target models.

Jouault and Tisi [21] propose an incremental technique for ATL that is based on creating bindings of OCL expressions to model elements. The bindings represent dependency information between specific rules and OCL expressions evaluated during the execution of the rules. With the dependency information, when changes are applied to the source model, exact rules which consume the modified model elements can be determined, and re-executed on such elements. This technique is different to the ones described above in that it relies on tracking transformation execution information. However, it exhibits a crucial limitation. It does not track imperative statements. As such, model elements accessed within imperative blocks can not be monitored. This compromises the correctness of the transformations that are executed with this incremental engine.

Of the reviewed incremental M2M transformation techniques [15, 16, 18] are purely declarative in nature. As such they are not applicable to transformations that require complex operations. On the other hand, [17, 20, 21] combine declarative and imperative constructs. However, they do not support incremental execution of imperative parts, instead [20] and [17] re-execute all imperative parts, hence, they are overly pessimistic and limit the benefits of incrementality. On the other hand [21] ignores imperative parts, thus it compromises the completeness of the transformation. On top of these limitations, using a M2T language to express a M2T transformation is typically awkward as M2M languages lack effective support for common M2T activities such as handling protected regions (to enable mixing generated with hand-written code), "static" (parts of generated) files, white space etc.

To the best of our knowledge, Xpand[10] is the only contemporary M2T language that supports source incremental transformation. Incremental generation in Xpand uses a combination of trace links and model differencing techniques. Difference models are used to determine changed subset of input models, and trace links are used to specify how source model elements are mapped to generated files. Once the difference model is constructed, impact analysis is performed to determine which changed model elements are used in which templates. A template is re-executed if it consumes a model element that has changed. The efficiency of the approach to incrementality employed by Xpand is heavily dependent on the effectiveness of the underlying modelling framework in performing model differencing. For instance, calculating model diffs between all the versions of GmfGraph models used for the Pongo transformation took about 1.3 seconds (average) using EmfCompare which is the same tool that Xpand uses to compute model diffs. Note that the time taken to compute the model diffs in this way represents only a part of the computation done by Xpand's incremental engine and exceeds the time taken to execute each Pongo transformation (see Table 2) on all versions of the GmfGraph model. As model differencing is integral to Xpand's incremental method, there is no need to conduct a full scale comparison of *property access traces* and model differencing incremental approaches.

[10] `http://eclipse.org/modeling/m2t/?project=xpand`

Furthermore, as recognised by the developers of Xpand[11], performance can be impaired because model differencing requires that (at least) two versions of the input model, along with a diff model are loaded, which requires at least three model traversals. This might also be impractical since access to the previous version of the model is needed and may not be available. Property access traces as explained in section 3 do not require model differencing and hence offer a fundamentally different approach to source incrementality.

6 Conclusions

In this paper, we proposed *property access traces*, an approach to reducing the execution time of M2T transformations in response to changes to source models. We have contributed a design for extending M2T transformation languages with support for *property access traces*, and demonstrated the feasibility of *property access traces* through an empirical evaluation. We have shown that the potential performance gains of source incremental transformation via property access traces are substantial: we observed an average reduction in transformation execution time of 60% and 75% for offline and online modes respectively. Instead of computing model differences between versions of input models as used by Xpand's incremental transformation technique, *property access traces* employs a technique that only requires the current state of a model.

In future work, we will investigate strategies for decomposing large monolithic templates that do not lend themselves to incrementality. By breaking monolithic templates (such as the one contained in the INESS transformation experiment (4.1.3)) into smaller units and by allowing multiple templates to contribute to the content of the same output file, such transformations can benefit from incremental techniques. We also intend to assess whether the use of different types of persistence mechanisms, such as a graph database or a triple store, can improve performance and/or lower space/memory requirements over our existing implementation which uses a relational database to store property access traces. Finally, we plan to implement support for integrating the incremental EGL engine with continuous integration systems such as Jenkins and Travis, so that the latter can trigger incremental re-execution of M2T transformations upon relevant model file change events.

Acknowledgements. This work was partially supported by the European Commission, through the Scalable Modelling and Model Management on the Cloud (MONDO) FP7 STREP project (grant #611125). The motivating example discussed in this paper was taken from Rose's work on the INESS project, which was supported by the European Commission and co-funded under the 7th Framework Programme (grant #218575).

[11] `http://help.eclipse.org/staging/neon2/index.jsp?topic=/org.eclipse.xpand.doc/help/incrementalGeneration_usage.html`

References

1. P. Mohagheghi et al. MDE adoption in industry: challenges and success criteria. In *Models in Software Engineering*, volume 5421, pages 54–59. Springer, 2009.
2. Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. Scalability: The holy grail of Model Driven Engineering. In *ChaMDE 2008 Workshop Proceedings*, pages 10–14, 2008.
3. Babajide Ogunyomi, Louis M. Rose, and Dimitrios S. Kolovos. Property Access Traces for Source Incremental Model-to-Text Transformation. In *Modelling Foundations and Applications - 11th European Conference, Held as Part of STAF, L'Aquila, Italy, July 20-24.*, pages 187–202, 2015.
4. Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. In *OOPSLA '03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.
5. K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.
6. B. Ogunyomi. Incremental model-to-text transformation (qualifying dissertation). Technical report, 2013.
7. Louis M Rose, Richard F Paige, Dimitrios S Kolovos, and Fiona AC Polack. The Epsilon Generation Language. In *Proc. ECMDA-FA*, volume 5095 of *LNCS*, pages 1–16. Springer, 2008.
8. Alexander Egyed. Automatically Detecting and Tracking Inconsistencies in Software Design Models. *Software Engineering, IEEE Transactions on*, 37(2):188–204, 2011.
9. Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd, 2nd edition, 2016.
10. Dimitrios S Kolovos, Louis M Rose, Saad Bin Abid, Richard F Paige, Fiona AC Polack, and Goetz Botterweck. Taming EMF and GMF using model transformation. In *Model Driven Engineering Languages and Systems*, pages 211–225. Springer, 2010.
11. Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: Eclipse Modeling Framework*. Pearson Education, 2nd edition, 2008.
12. Gerard J Holzmann. *The SPIN model checker: Primer and reference manual*, volume 1003. Addison-Wesley Reading, 2004.
13. Jan Friso Groote, Aad Mathijssen, Michel Reniers, Yaroslav Usenko, and Muck Van Weerdenburg. *The Formal Specification Language mCRL2*. Citeseer, 2007.
14. Babajide Ogunyomi, Louis M Rose, and Dimitrios S Kolovos. On the use of signatures for source incremental model-to-text transformation. In *MoDELS*, volume 8767 of *LNCS*, pages 84–98. Springer, 2014.
15. David Hearnden, Michael Lawley, and Kerry Raymond. Incremental model transformation for the evolution of model-driven systems. In *Proc. MoDELS*, LNCS, pages 321–335. Springer, 2006.
16. Holger Giese and Robert Wagner. Incremental model synchronization with triple graph grammars. In *Model Driven Engineering Languages and Systems*, pages 543–557. Springer, 2006.
17. István Ráth, Gábor Bergmann, András Ökrös, and Dániel Varró. Live model transformations driven by incremental pattern matching. In *Theory and Practice of Model Transformations*, pages 107–121. Springer, 2008.
18. Holger Giese, Stephan Hildebrandt, and Stefan Neumann. Model synchronization at work: keeping SysML and AUTOSAR Models Consistent. In *Graph Transformations and Model-driven Engineering*, pages 555–579. Springer, 2010.

19. Charles L Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial intelligence*, 19:17–37, 1982.
20. Bergmann, Gábor and Ráth, István and Varró, Gergely and Varró, Dániel. Change-driven Model Transformations. *Software & Systems Modeling*, 11(3):431–461, 2012.
21. Frédéric Jouault and Massimo Tisi. Towards incremental execution of ATL transformations. In *Theory and Practice of Model Transformations*, pages 123–137. Springer, 2010.